METHOD AND APPARATUS FOR CONTROLLING EXECUTION OF A CHILD PROCESS GENERATED BY A PARENT PROCESS BEING MONITORED UNDER AN INSTRUMENTATION SCHEME

BACKGROUND

[0001] The computing community has developed tools and methods to analyze the run-time behavior of a computer program. Many of the tools and methods use statistical sampling and binary instrumentation techniques. Statistical sampling is performed by recording periodic snapshots of the program's state, e.g., the program's instruction pointer. Sampling imposes a low overhead on a program's run time performance, is relatively non-intrusive, and imprecise. For example, a sampled instruction pointer may not be related to the instruction address that caused a particular sampling event.

[0002] While binary instrumentation leads to more precise results, the accuracy comes at some cost to the run-time performance of the instrumented program. Because the binary code of a program is modified, all interactions with the processor and the operating system can change significantly. For example, additional instructions and changes to a program's cache and paging behaviors can cause run-time performance increases from a few percent up to 400%. Consequently, binary instrumentation is considered intrusive.

[0003] Dynamic binary instrumentation allows program instructions to be changed on the fly and leads to a whole class of more precise run-time monitoring results. Unlike static binary instrumentation techniques that are applied over an entire program prior to execution of the program. Dynamic binary instrumentation is performed at run-time of a program and only instruments those portions of an executable that are executed. Consequently, dynamic binary instrumentation techniques can significantly reduce the overhead imposed by the instrumentation process.

[0004] Software development tools can combine statistical sampling and dynamic binary instrumentation methods into a framework that enables performance analysis, profiling, coverage analysis, correctness checking, and testing of a program.

[0005] A basic reason for the difficulty in testing the correctness of a program is that program behavior largely depends on the data on which the program operates and, in case of interactive programs, on the information (data and commands) received from a user. Therefore even if exhaustive testing is impossible, as is often the case, program testing and verification is preferably conducted by causing the program to operate with some data. In other words, by creating and performing what is defined as a "process" by software designers.

[0006] A "process" is commonly defined as an address space, a control thread operating within the address space, and the set of system resources needed for operating with the thread. Therefore, a "process" is a logic entity consisting of the program itself, the data on which it must operate, the memory resources, and input / output resources. Program verification and performance testing encompasses execution of the program as a process thread to test if the process develops in the correct way or if undesired or unexpected events occur.

[0007] Generally, software development tools use two basic techniques to controllably execute program instructions, tracing functions, or tracers and symbolic analysis functions, or symbolic debuggers.

[0008] Tracing functions modify a program to be tested so that select program instructions are preceded and followed by overhead instructions that extract variable information, control execution of the instruction, and can monitor program execution. Symbolic debuggers are interactive programs, which translate a high-level language source program to be tested into a compiled program. Symbolic debuggers modify an executable copy of the source selectively inserting conditional branches to other routines, instruction sequences, and break points. The compiled and instrumented program can then be run under the control of a managing program or a software engineer via a human machine interface.

[0009] Symbolic debuggers also enable the insertion of instruction sequences for recording variables used in execution of the instruction and, on user request, can add and remove break points, modify variables, and permit modification of the hardware environment. These techniques are particularly effective in that they permit step by

step control of the execution of a program, that is, they allow the evolution of the related process to be controlled by halting and restarting the process at will and by changing parameters during the course of execution of the process. The tools also can display the execution status of the process to the software engineer in detail by means of display windows or other output devices that enable the user to continuously monitor the program. Some tools automate the process of setting break points in the executable version of the source code.

[0010] Symbolic debuggers have several limitations. First, they operate on only a single process at a time. Second, the process to be tested must be activated by the parent process (the symbolic analysis process) and cannot be activated earlier. Consequently, the debugging of programs, which are activated at system start up, such as monitors, daemons, etc., is problematic.

[0011] Furthermore, because the process to be tested is generated as a child of the symbolic analysis parent, and in a certain sense is the result of a combination of the symbolic analysis function/program with the program to be tested, the two processes must share or utilize the same resources. As a consequence, interactive programs that use masks and windows on a display device cannot be tested because they compete or interfere with the symbolic debugger in requiring access to the display device.

[0012] Moreover, software development tools that use symbolic debuggers can encounter deadlock conditions that result from the standard execution of operating system level instructions. One operating system that has gained widespread acceptance is the UNIX® operating system. UNIX® is a trademark of the American Telephone and Telegraph Company of New York, New York, U.S.A.

[0013] The UNIX® operating system is a multi-user, time-sharing operating system with a tree-structured file system. Other noteworthy functional features are its logical I/O capabilities, pipes, and forks. The logical I/O capabilities allow a user to specify the input and output files of a program at runtime rather than at compile time, thus providing greater flexibility. Piping is a feature that enables buffering of input and output data to and from other processes. Forking is a feature that enables the creation of a new process.

[0014] By themselves, these features offer no inherent benefits. However, the UNIX® operating system command environment (called the SHELL) provides easy access to these operating system capabilities and also allows them to be used in different

combinations. With the proper selection and ordering of system commands, logical I/O, pipes, and forks, a user at the command level can accomplish tasks that on other operating systems would require writing and generating an entirely new program. This ability to easily create application program equivalents from command level is one of the unique and primary benefits of the UNIX® operating system.

[0015] The popularity of the UNIX® operating system has led to the creation of numerous open source and proprietary variations such as LINUX®, HP-UX®, PRIMIX®, etc. LINUX® is a trademark of William R. Della, Jr. (individual) of Boston, Massachusetts, U.S.A. HP-UX®, is a trademark of the Hewlett-Packard Company, of Palo Alto, California, U.S.A. PRIMIX® is a trademark of Primix Solutions, Inc., of Watertown, Massachusetts, U.S.A. These variants of the UNIX® operating system inherently use the UNIX® operating system's logical I/O capabilities, pipes, and forks.

[0016] Software development tools can encounter a deadlock condition when a process under test includes a "fork" instruction. The operation of a "fork" instruction in the UNIX® operating system involves spawning a new process, and then copying the process image of the parent (the process making the fork call) to the child process (the newly spawned process).

[0017] FIG. 1 illustrates the deadlock condition. Deadlock condition 10 occurs between development tool 20, parent process 30, and child process 40 as follows. Development tool 20 instruments parent process 30 as indicated in block 22. Development tool 20 assigns a process identifier (process ID) to the parent process 30 in block 24. Next, the development tool 20 monitors execution of the parent process using trace control as shown in block 26. Under the UNIX® operating system and its open source and proprietary variants, development tool 20 waits for trace events that include the process ID of the parent process as indicated in block 28. Development tool 20 cannot monitor child process 40, since child process 40 has not been created.

[0018] Once parent process 30 is created and started, parent process 30 runs nominally in accordance with its instructions until it encounters a fork instruction (e.g., fork or vfork) as shown in block 32. Thereafter, as shown in block 34, parent process 30 assigns a process ID, different from its own process ID, to identify the child process. In accordance with the fork instruction, parent process 30 copies itself in its instrumented state to spawn child process 40 and generates a trace event which

is received by development tool 20. Thereafter, as shown in block 38, parent process 30 is essentially suspended waiting for an indication that child process 40 has completed (e.g., indicia of an exec or exit).

[0019] Once child process 40 is created by the fork instruction in parent process 30, child process 40 runs nominally in accordance with its instructions until it encounters the fork instruction shown in block 42. Thereafter, as shown in block 44, child process 40 assigns a process ID, different from its own process ID, to identify the subsequent child process. As illustrated in block 46, in accordance with the fork instruction, child process 40 copies itself in its instrumented state to spawn the subsequent child process (not shown) and generates a trace event which is ignored by development tool 20 because development tool 20 is only looking for trace events from parent process 30.

[0020] Once the fork instruction is encountered and processed in child process 40, the deadlock condition has occurred. Parent process 30 is suspended waiting for an indication that child process 40 has completed. Child process 40, which inherited trace control from parent process 30, waits for a process to handle the trace event generated at the time it executed the fork instruction. Development tool 20 waits for a trace event from parent process 30.

[0021] Consequently it is desirable to have an improved apparatus, program, and method for avoiding fork instruction induced deadlocks when using debugging techniques to instrument and monitor computer programs.

SUMMARY

[0022] An embodiment of a debug interface, includes a pre-fork event responsive to a fork instruction call wherein the pre-fork event includes indicia that identifies a child process to be created in accordance with the fork instruction call.

[0023] An embodiment of a method for controlling the execution of a child process created from a parent process, where the parent process is instrumented by a software tool includes, receiving indicia that a fork instruction will be executed by the parent process, suspending execution of the parent process, extracting a process identifier from the indicia of the fork instruction, the process identifier corresponding to a child process to be generated by the parent process when the parent process executes the fork instruction, setting a process monitor thread to observe the child process, and

resuming execution of the parent process to enable the parent process to execute the fork instruction.

BRIEF DESCRIPTION OF THE DRAWINGS

[0024] Systems and methods for controlling execution of a child process generated by an instrumented parent process are illustrated by way of example and not limited by the implementations in the following drawings. The components in the drawings are not necessarily to scale, emphasis instead is placed upon clearly illustrating the principles used in controlling the execution of such a child process. Moreover, in the drawings, like reference numerals designate corresponding parts throughout the several views.

[0025] FIG. 1 is a composite flowchart illustrating a prior art deadlock condition.

[0026] FIG. 2 is a functional block diagram of a computing device.

[0027] FIG. 3 is a functional block diagram of an embodiment of a software monitor.

[0028] FIG. 4 is a flow chart illustrating an embodiment of a method for controlling execution of a child process that can be implemented by the software monitor of FIG. 3.

[0029] FIG. 5 is a flow chart illustrating an embodiment of a method for executing a parent process that can be implemented by the software monitor of FIG. 3.

[0030] FIG. 6 is a flow chart illustrating an embodiment of a method for executing a process monitor that can be implemented by the software monitor of FIG. 3.

[0031] FIG. 7 is a flow chart of an embodiment of a method for controlling the execution of a child process.

[0032] FIG. 8 is a flow chart of an embodiment of a method for executing an instrumented parent process to ensure execution of a child process.

DETAILED DESCRIPTION

[0033] The improved debug interface uses system calls to monitor or otherwise control thread and process execution. A thread is that part of a program that can execute independently of other parts of the program. Operating systems such as UNIX® that support multithreading, enable programmers to design programs whose threaded parts can execute concurrently.

[0034] Tracing facilities enable a process or thread to control the execution of another process or thread, respectively. Tracing facilities enable breakpoint and event driven

debugging of programs. Thread trace or ttrace is a tracing facility for single and multithreaded processes. Process trace or ptrace is a tracing facility that enables a parent process to manipulate the state of a cooperative child process. While under the control of a tracing facility, the traced code behaves normally until one of its threads or the process encounters a signal, or an event. When the signal or event is encountered, the thread or process enters a stopped or suspended state and the tracing process is notified of the signal or the event via a ttrace_wait instruction. The instruction includes an argument that determines the action to be taken.

[0035] A tracing process can set event flags in the context of a traced process, or its individual threads, to cause the threads to respond to specific events during their execution. When an event flag is set in the context of the process, all threads in the process respond to the event. When set in the context of a thread, only the specific thread will respond to the event.

[0036] If an event is requested by a process, the event mask of the thread is not examined. For the event mask of the thread to be significant, the process event must be unset. Similarly, if an event option is enabled in the process, the option for the thread is not considered. Event masks may be inherited across fork instructions. For example, if tteo_proc_inherit is set, the child process inherits the event mask of its parent. By default, threads stop when they receive a signal. If the signal being processed has its mask bit set, signal processing continues as though the process was not being traced. The traced thread is not stopped, and the tracing process is not notified of the signal. On the other hand, if the signal mask bit is not set for the signal being processed, the traced thread is stopped and the tracing process is notified via ttrace_wait.

[0037] As explained above, fork instructions (i.e., fork and vfork) present a deadlock condition for debuggers that use known tracing facilities. For example, when the ttevt_fork event flag is set under the ttrace tracing facility both the parent thread and the initial thread in the child process stop (after the child process is marked as a traced process and adopts the parent thread's debugger). Both threads log the fact that they stopped in response to a ttevt_fork event. In the case of a vfork instruction where the ttev_vfork event flag is set, when the child process stops, its parent is asleep, and the child borrows the parent's address space until a call to exec or an exit (either by a call

to exit or an abnormal termination of the child) takes place. Consequently, continuing the parent process before the child has completed results in an error.

[0038] In response, the improved debug interface includes a pre-fork event and associated processing that can be adapted to multiple tracing facilities such as ttrace and/or ptrace. The improved debug interface and the associated methods described below enable a software tool to control the execution of a child process initiated by an instrumented parent process, where the parent process includes one or more fork instructions.

[0039] Turning now to the drawings, wherein like-reference numerals designate corresponding parts throughout the drawings; reference is made to FIG. 2, which illustrates a functional block diagram of a computing device. Generally, in terms of hardware architecture, as shown in FIG. 2, computing device 200 may include a processor 210, memory 220, input/output device interface(s) 230, and LAN/WAN interface(s) 240 that are communicatively coupled via local interface 250. The local interface 250 can be, for example but not limited to, one or more buses or other wired or wireless connections, as known in the art or that may be later developed. Local interface 250 may have additional elements, which are omitted for simplicity, such as controllers, buffers (caches), drivers, repeaters, and receivers, to enable communications. Further, local interface 250 may include address, control, and/or data connections to enable appropriate communications among the aforementioned components.

[0040] In the embodiment of FIG. 2, the processor 210 is a hardware device for executing software that can be stored in memory 220. The processor 210 can be any custom-made or commercially available processor, a central processing unit (CPU) or an auxiliary processor among several processors associated with the computing device 200, a semiconductor-based microprocessor (in the form of a microchip) or a macroprocessor.

[0041] Memory 220 can include any one or combination of volatile memory elements (e.g., random access memory (RAM, such as dynamic RAM or DRAM, static RAM or SRAM, etc.)) and nonvolatile memory elements (e.g., read-only memory (ROM), hard drives, tape drives, compact discs (CD-ROM.). Moreover, the memory 220 may incorporate electronic, magnetic, optical, and/or other types of storage media now known or later developed. Note that memory 220 can have a distributed architecture,

where various components are situated remote from one another, but accessible by processor 210.

[0042] The software in memory 220 may include one or more separate programs, each of which comprises an ordered listing of executable instructions for implementing logical functions. In the example of FIG. 2, the software in the memory 220 includes operating system 222, one or more application(s) 224, and a software monitor 300. Application(s) 224 and software monitor 300 function as a result of and in accordance with operating system 222. Operating system 222 controls the execution of the other application(s) 224 and computer programs, such as software monitor 300, and provides scheduling, input-output control, file and data management, memory management, and communication control and related services.

[0043] Software monitor 300 and application(s) 224 include one or more source programs, executable programs (object code), scripts, or other collections each comprising a set of instructions to be performed. As will be explained in detail below, software monitor 300 includes logic that controls the execution of application(s) 224. More specifically, software monitor 300 includes logic that controls the execution of a child process or thread generated by an instrumented parent process found within application(s) 224 where the parent process or thread includes a fork instruction. It should be well-understood by one skilled in the art, after having become familiar with the teachings of the improved debug interface, that software monitor 300 and application(s) 224 may be written in a number of programming languages now known or later developed. Moreover, software monitor 300 and application(s) 224 may be stored across distributed memory elements in contrast with memory 220 shown in FIG. 2.

[0044] The input/output device(s) 230 may take the form of human/machine devices, such as but not limited to, a keyboard, a mouse or other suitable pointing device, a microphone, etc. Furthermore, the input/output device(s) 230 may also include known or later developed input/output devices, for example but not limited to, a printer, a display device, an external speaker, etc.

[0045] Network-interface device(s) 240 may include a host of devices that may establish one or more communication sessions between computing device 200 and one or more local and/or wide area networks. Network-interface device(s) 240 may include but are not limited to, a modulator/demodulator or modem (for accessing

another device, system, or network); a radio frequency (RF) or other transceiver; a telephonic interface; a bridge; an optical interface; a router; etc. For simplicity of illustration and explanation, these aforementioned two-way communication devices are not shown.

[0046] When the computing device 200 is in operation, the processor 210 is configured to execute software stored within the memory 220, to communicate data to and from the memory 220, and to generally control operations of the computing device 200 pursuant to the software. Operating system 222, one or more application(s) 224, and the software monitor 300, in whole or in part, but typically the latter, are read by the processor 210, perhaps buffered within the processor 210, and then executed in accordance with the respective instructions.

[0047] It should be understood that software monitor 300 can be embodied in any computer-readable medium for use by or in connection with an instruction execution system, apparatus, or device, such as a computer-based system, processor-containing system, or other system that can fetch the instructions from the instruction execution system, apparatus, or device, and execute the instructions. A "computer-readable medium" can be any methods and resources for storing, communicating, propagating, or transporting a program for use by or in connection with the instruction execution system, apparatus, or device. The computer-readable medium can be, for example but not limited to, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, apparatus, device, or propagation medium now known or later developed. Note that the computer-readable medium could even be paper or another suitable medium upon which the program is printed, as the program can be electronically captured, via for instance optical scanning of the paper or other medium, then compiled, interpreted or otherwise processed in a suitable manner if necessary, and then stored in a computer memory.

[0048] Those skilled in the art will understand that various portions of software monitor 300 can be implemented in hardware, software, firmware, or combinations thereof. In a preferred embodiment, software monitor 300 is implemented using software that is stored in memory 220 and executed by a suitable instruction execution system. If implemented solely in hardware, as in an alternative embodiment, software monitor 300 can be implemented with any or a combination of technologies well-known in the art (e.g., discrete logic circuits, application specific integrated circuits

(ASICs), programmable gate arrays (PGAs), field programmable gate arrays (FPGAs), *etc.*), or technologies later developed.

[0049] In a preferred embodiment, the software monitor 300 is implemented via a combination of software and data stored in memory 220 and executed and stored or otherwise processed under the control of processor 210. It should be noted, however, that software monitor 300 is not dependent upon the nature of the underlying processor 210 or memory 220 in order to accomplish designated functions.

[0050] Reference is now directed to FIG. 3, which presents a functional block diagram of an embodiment of software monitor 300. As illustrated in FIG. 3, software monitor 300 comprises an instrumentation engine 310, a process monitor 320, and a debug interface 330. Before software monitor 300 can collect and analyze performance information regarding a specific thread or process, instrumentation engine 310 inserts code into the target process or thread. Preferably, software monitor 300 contains logic that in accordance with dynamic binary-instrumentation techniques, instruments only those portions of parent process 350 that will be executed by processor 210.

[0051] Instrumentation engine 310 may receive data via various input/output devices 230, data stored in memory 220 (FIG. 2), as well as various application(s) 224. The data will identify one or more target processes or threads to instrument. In addition, the data may include various parameters and flags that instrumentation engine 310 uses in generating parent process 350. Alternatively, instrumentation engine 310 can be programmed with one or more default parameters to apply when instrumenting the target process. Instrumentation engine 310, having received data identifying the target process or thread applies the various parameters and flags and generates parent process 350. Parent process 350 is an instrumented version of the identified target process or thread.

[0052] As further illustrated in the functional block diagram of FIG. 3, software monitor includes process monitor 320. Process monitor 320 includes logic for coordinating the collection of data during execution of parent process 350. When parent process 350 includes one or more fork instructions, process monitor 320 ensures that data collected during execution of both the parent process 350 and its child process 352 are associated with the process responsible for generating the data. As shown in FIG. 3, process monitor 320 functions through debug interface 330 and the underlying operating system

222. Information flows between process monitor 320 and debug interface 330 include trace system calls, events, and signals 332.

[0053] Debug interface 330 includes logic for receiving and responding to the various trace system calls, events, and signals 332. In addition, debug interface 330 includes logic for generating instructions 335. Instructions 335 are in accordance with the underlying operating system 222. As illustrated in FIG. 3, debug interface 330 receives and responds to trace system calls, events, and signals 332 generated and sent by parent process 350, child process 352, and process monitor 320.

[0054] Operating system 222 as illustrated in FIG. 3, sends and receives instructions 335 both to and from debug interface 330. In addition, operating system 222 receives a fork instruction 334 (e.g., fork or vfork) from parent process 350. As described in the UNIX® operating system and many of its proprietary and open source derivatives, fork instruction 334 suspends execution of parent process 350 and generates child process 352 which contains a copy of the instrumented code and trace calls, events, and signals contained within parent process 350.

[0055] However, in addition to the other trace system calls, events, and signals 332, parent process 350 communicates a pre-fork event 355 to debug interface 330. Pre-fork event 355 includes indicia identifying child process 352 before it is created by a subsequently executed fork instruction within parent process 350. The indicia includes at least a process identifier of the child process 352. Debug interface 330 further includes logic configured to recognize and respond to the pre-fork event 355.

[0056] While the functional block diagram presented in FIG. 3 illustrates software monitor 300 as having a single centrally-located instrumentation engine 310 with co-located process monitor 320 and debug interface 330, it should be understood that the various functional elements of software monitor 300 may be distributed across multiple locations in memory 220 and/or across multiple memory devices (not shown). It should be further understood that instrumentation engine 310 is not limited to dynamic binary instrumentation techniques and may include logic in accordance with binary instrumentation techniques (i.e., logic that instruments all portions of the identified parent process 350) and statistical sampling.

[0057] Reference is now directed to the flow chart illustrated in FIG. 4. In this regard, the various functions shown in the flow chart present a method for controlling the execution of a child process created from a parent process, where the parent process is

instrumented by a software tool that may be realized by software monitor 300. As illustrated in FIG. 4, the method may begin by determining whether it is desired to instrument a parent process as shown in query 402. When it is determined that it is desirable to instrument the parent process as indicated by the flow control arrow labeled "YES" that exits query 402, the method responds by acquiring desired monitoring parameters, as indicated in block 404. Otherwise, as indicated by the flow control arrow labeled "NO" the method for controlling the execution of a child process created from a parent process can be bypassed.

[0058] Returning to the condition where it is desired to instrument the parent process, as illustrated in block 406, a process monitor thread is set to communicate with the parent process. As shown in block 408, the parent process is instrumented and executed.

[0059] Thereafter, as indicated in the wait loop formed by query 410 and wait block 412, execution of the parent process continues until a pre-fork event is detected. When a pre-fork event is detected, as indicated by the flow control arrow labeled "YES" exiting query 410, the function indicated in block 414 is performed. More specifically, the process identifier of the child process that will be created by the subsequent fork instruction is extracted from the pre-fork event. Thereafter, as indicated in block 416, a process monitor thread is configured to respond to trace events generated by the future child process. Next, as shown in block 418, execution of the parent process resumes to permit the parent process to execute the subsequent fork instruction.

[0060] As described above, a child process that contains a copy of the parent process (including code inserted by an instrumentation process) is generated and executed in accordance with the fork instruction. In addition, the parent process is suspended until the child process terminates (e.g., the child process generates an exec or exit event). When it is determined that the child process has terminated, as indicated by the flow control arrow labeled "YES" exiting query 420, the process monitor is configured to monitor trace events generated by the parent process as indicated in block 424 before resuming execution of the parent process.

[0061] Execution of the parent process continues, as indicated in the wait loop formed by query 426 and wait block 428, until the parent process terminates (e.g., the parent process generates an exec or exit event). When it is determined that the parent process has terminated, as indicated by the flow control arrow labeled "YES" exiting query 426,

the process monitor is suspended as indicated in block 430 and run-time data observed during execution of the parent process is collected and analyzed as shown in block 432.

[0062] Those skilled in the art will understand that the method for controlling the execution of a child process created from a parent process illustrated in FIG. 4 is configured to successfully control a single child process generated as the result of the execution of the first fork instruction within an instrumented parent process. A software monitor may include multiple threads for controlling the performance of the desired functions. For example, a first thread may continuously wait for and process pre-fork events. A parallel (i.e., simultaneously executed) thread may continuously wait for an indication that the presently active process has terminated. These parallel threads may be executed as may be desired by a software monitor.

[0063] FIG. 5 is a flow chart illustrating an embodiment of a method for executing a parent process instrumented by a software tool to ensure execution of a child process when the parent process contains a fork instruction. As illustrated in query 502, the parent process 350 begins by determining if a fork or vfork instruction is about to be executed by the parent process or thread. When it is determined that a fork or vfork instruction is about to be executed by the parent process or thread as indicated by the flow control arrow labeled "YES" that exits query 502, the parent process generates a pre-fork event as indicated in block 504. Next, as shown in block 506, the parent process sends the pre-fork event to the software tool responsible for instrumenting the parent.

[0064] Thereafter, as indicated in the wait loop formed by query 508 and wait block 510, execution of the parent process is suspended until after the parent receives an indication from the software tool that the pre-fork event has been successfully processed. When the pre-fork event has been processed, as indicated by the flow control arrow labeled "YES" exiting query 508, the parent is activated and executes the fork instruction as shown in block 512. Once the fork instruction has been executed, the parent is suspended as indicated in block 514.

[0065] As indicated in the wait loop formed by query 516 and wait block 518, the parent process remains suspended until the parent receives an indication that the child process has terminated (e.g., the child process generates an exec or an exit event). When the child process has terminated, as indicated by the flow control arrow labeled "YES" exiting query 516, the parent process resumes as shown in block 520. As indicated in

query 522, the parent process continues until it terminates nominally and sends an exec event or fails and sends an exit event. As shown by the flow control arrow labeled "NO" exiting query 522, the parent is configured to report any future fork or vfork instructions by repeating the functions and queries described above.

[0066] Those skilled in the art will understand that the method for executing a parent process instrumented by a software tool to ensure execution of a child process when the parent process contains a fork instruction illustrated in FIG. 5 is configured to generate and send a pre-fork event before executing a fork or vfork instruction. The parent process or thread may be implemented via multiple threads for controlling the performance of the desired functions. For example, a first thread may continuously identify when a fork instruction is encountered in the instruction sequence. A second thread may intermittently be started to wait for an indication that the software tool has successfully processed the pre-fork event. A third thread may be responsible for handling trace events generated by the child process. These and other threads may be executed as may be desired by a parent process or thread to implement the various functions illustrated in FIG. 5.

[0067] Reference is now directed to the flow chart illustrated in FIG. 6, which illustrates an embodiment of a method for controllably switching a target process of a process monitor thread between an instrumented parent process and a child process generated by the parent process. In this regard, process monitor 320 begins with query 602 where it is determined if the child process has been successfully generated and started. If the result of query 602 indicates that the child process has not started successfully, process monitor 320 is configured to wait as indicated in block 604. Next, query 606 is performed to determine if the parent process has received an indication that the fork instruction failed. When the parent process receives an indication that the fork instruction failed as indicated by the flow control arrow labeled "YES," the process monitor sets the active process identifier (PID) to the parent process' PID as shown in block 608. Otherwise, the process monitor returns to query 602. The determinations made in query 602 and query 604 are repeated to handle the case where an event documenting the creation of the child process as a result of the fork instruction has not been received before the process monitor is started.

[0068] After the process monitor has set the PID to the parent process' PID, the process monitor continues by monitoring events and signals generated by the parent process as indicated by the monitoring loop formed by query 616 and block 614.

[0069] When the result of query 602 indicates that the child process has started successfully, process monitor 320 is configured to perform query 610 to determine if the parent process received an indication that the fork instruction failed. In this way, the process monitor confirms that a child process was not generated by the parent process with the same PID identified in the pre-fork event. When the result of query 610 indicates that the fork instruction failed, the process monitor is configured to notify the software monitor 300 that the parent process has started two processes with the same PID as indicated in block 612. Otherwise, when query 610 indicates that the fork instruction has not failed, the process monitor continues by executing the monitoring loop formed by query 616 and block 614. When query 602 indicates that the child process has started successfully and query 610 indicates that the parent process has not received an indication that the fork instruction has failed, the target PID will reflect the PID of the child process generated by the fork instruction executed by the parent process.

[0070] FIG. 7 is a flow chart of an embodiment of a method 700 for controlling the execution of a child process. When an appropriately configured software tool receives indicia that a fork instruction is about to be executed by a parent process, as indicated in block 702, the software tool suspends execution of the parent process as illustrated in block 704. While the parent process is suspended, as indicated in block 706, the software tool extracts a process identifier from the indicia of the fork instruction corresponding to a child process to be generated by the process when the parent executes the pending fork instruction. In block 708, the software tool sets a process monitor thread to observe the execution of the child process. The software tool then resumes execution of the parent process as indicated in block 710.

[0071] FIG. 8 is a flow chart of an embodiment of a method for executing an instrumented parent process to ensure execution of a child process. When an appropriately configured software tool receives indicia that a fork instruction is about to be executed by a parent process, as indicated in block 802, the software tool generates a pre-fork event that includes indicia of a child process that will be generated by the fork instruction, as shown in block 806. Otherwise, the software tool waits for an amount of

time to pass, as illustrated by the flow control arrow labeled "No" exiting block 802 and wait block 804 before repeating the query of decision block 802.

[0072] Once the pre-fork event has been generated, the software tool receives the pre-fork event as shown in block 808. Thereafter, as indicated in decision block 810, the software tool determines if the pre-fork event has processed successfully. When it is determined that the pre-fork event has not completed successfully, as indicated by the flow control arrow labeled "No" exiting block 810 and wait block 812, the software tool repeats the query of decision block 810. It should be understood that this portion of the flow chart assumes that the pre-fork event will eventually terminate successfully. Those skilled in the art may insert additional failure handling mechanisms to recover from a pre-fork instruction execution failure.

[0073] Otherwise, when it is determined that the pre-fork event has successfully terminated, as indicated by the flow control arrow labeled "Yes" exiting decision block 810, the software tool executes the fork instruction as shown in block 814. Thereafter execution of the parent process is suspended as shown in block 816 to enable the software tool to monitor execution of the child process.

[0074] Any process descriptions or blocks in the flow charts presented in FIGs. 4-8 should be understood to represent modules, segments, or portions of code or logic, which include one or more executable instructions for implementing specific logical functions in the associated process. Alternate implementations are included within the scope of the disclosed methods in which functions may be executed out of order from that shown or discussed, including substantially concurrently or in reverse order, depending on the functionality involved, as would be understood by those reasonably skilled in the art after having become familiar with the improved debug interface and the associated methods for controlling the execution of a child process generated by an instrumented parent process.